Part II
Circuit-Level Parallelism

- Sorting and Selection Networks
- Search Acceleration Circuits
- Arithmetic and Counting Circuits
- Fourier Transform Circuits
About This Presentation

This presentation is intended to support the use of the textbook *Introduction to Parallel Processing: Algorithms and Architectures* (Plenum Press, 1999, ISBN 0-306-45970-1). It was prepared by the author in connection with teaching the graduate-level course ECE 254B: Advanced Computer Architecture: Parallel Processing, at the University of California, Santa Barbara. Instructors can use these slides in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Spring 2005</td>
<td>Spring 2006</td>
<td>Fall 2008</td>
<td>Fall 2010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Winter 2019</td>
<td>Winter 2020</td>
<td></td>
</tr>
</tbody>
</table>
II” Circuit-Level Parallelism

Circuit-level specs: most realistic parallel computation model
- Concrete circuit model; incorporates hardware details
- Allows realistic speed and cost comparisons
- Useful for stand-alone systems or acceleration units

<table>
<thead>
<tr>
<th>Topics in This Part</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 7</td>
<td>Sorting and Selection Networks</td>
</tr>
<tr>
<td>Chapter 8A</td>
<td>Search Acceleration Circuits</td>
</tr>
<tr>
<td>Chapter 8B</td>
<td>Arithmetic and Counting Circuits</td>
</tr>
<tr>
<td>Chapter 8C</td>
<td>Fourier Transform Circuits</td>
</tr>
</tbody>
</table>
7 Sorting and Selection Networks

Become familiar with the circuit model of parallel processing:
- Go from algorithm to architecture, not vice versa
- Use a familiar problem to study various trade-offs

Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>What is a Sorting Network?</td>
</tr>
<tr>
<td>7.2</td>
<td>Figures of Merit for Sorting Networks</td>
</tr>
<tr>
<td>7.3</td>
<td>Design of Sorting Networks</td>
</tr>
<tr>
<td>7.4</td>
<td>Batcher Sorting Networks</td>
</tr>
<tr>
<td>7.5</td>
<td>Other Classes of Sorting Networks</td>
</tr>
<tr>
<td>7.6</td>
<td>Selection Networks</td>
</tr>
</tbody>
</table>
7.1 What is a Sorting Network?

The outputs are a permutation of the inputs satisfying $y_0 \leq y_1 \leq \cdots \leq y_{n-1}$ (non-descending).

Fig. 7.1 An $n$-input sorting network or an $n$-sorter.

Fig. 7.2 Block diagram and four different schematic representations for a 2-sorter.
Building Blocks for Sorting Networks

Fig. 7.3 Parallel and bit-serial hardware realizations of a 2-sorter.
Proving a Sorting Network Correct

Fig. 7.4  Block diagram and schematic representation of a 4-sorter.

Method 1: Exhaustive test – Try all $n!$ possible input orders

Method 2: Ad hoc proof – for the example above, note that $y_0$ is smallest, $y_3$ is largest, and the last comparator sorts the other two outputs

Method 3: Use the zero-one principle – A comparison-based sorting algorithm is correct iff it correctly sorts all 0-1 sequences ($2^n$ tests)
Elaboration on the Zero-One Principle

Let outputs $y_i$ and $y_{i+1}$ be out of order, that is $y_i > y_{i+1}$

Replace inputs that are strictly less than $y_i$ with 0s and all others with 1s

The resulting 0-1 sequence will not be correctly sorted either
### 7.2 Figures of Merit for Sorting Networks

<table>
<thead>
<tr>
<th>Cost</th>
<th>Delay</th>
<th>Cost x Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comparators</td>
<td>Number of levels</td>
<td>The following 4-sorter has 3 comparator levels on its critical path</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>The cost-delay product for this example is 15</td>
</tr>
</tbody>
</table>

In the following example, we have 5 comparators.

The following 4-sorter has 3 comparator levels on its critical path.

The cost-delay product for this example is 15.

![Block diagram and schematic representation of a 4-sorter.](image)

**Fig. 7.4** Block diagram and schematic representation of a 4-sorter.
Cost as a Figure of Merit

Optimal size is known for $n = 1$ to $8$: 0, 1, 3, 5, 9, 12, 16, 19

Fig. 7.5 Some low-cost sorting networks.

This figure has been updated from Fig. 49, p. 227, in the 1998 edition of Donald Knuth’s *The Art of Computer Programming, Vol. 3*
Delay as a Figure of Merit

Optimal delay is known for $n = 1$ to $10$: 0, 1, 3, 3, 5, 5, 6, 6, 7, 7

$n = 6$
12 modules, 5 levels

$n = 9$
25 modules, 7 levels
(This one is incorrect)

$n = 10$
31 modules, 7 levels

$n = 12$
40 modules, 8 levels

$n = 16$
61 modules, 9 levels

Fig. 7.6 Some fast sorting networks.

This figure has been updated from Fig. 51, p. 229, in the 1998 edition of Donald Knuth’s *The Art of Computer Programming*, Vol. 3
Best Sorting Networks Known

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth $^{[10]}$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Size, upper bound $^{[11]}$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>16</td>
<td>19</td>
<td>25</td>
<td>29</td>
<td>35</td>
<td>39</td>
<td>45</td>
<td>51</td>
<td>56</td>
<td>60</td>
<td>71</td>
</tr>
<tr>
<td>Size, lower bound (if different) $^{[11]}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td>37</td>
<td>41</td>
<td>45</td>
<td>49</td>
<td>53</td>
<td>58</td>
</tr>
</tbody>
</table>


References:


The problem of determining whether a given candidate network is a sorting network is co-NP-complete

Cost-Delay Product as a Figure of Merit

- Cost × Delay = 29 × 9 = 261 (Low-cost 10-sorger from Fig. 7.5)
- Cost × Delay = 31 × 7 = 217 (Fast 10-sorger from Fig. 7.6)

The most cost-effective n-sorger may be neither the fastest design, nor the lowest-cost design.
7.3 Design of Sorting Networks

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C(n)$</td>
<td>$n(n-1)/2$</td>
</tr>
<tr>
<td>$D(n)$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Cost × Delay = $n^2(n-1)/2 = \Theta(n^3)$

Fig. 7.7 Brick-wall 6-sorter based on odd–even transposition.
Insertion Sort and Selection Sort

Parallel insertion sort = Parallel selection sort = Parallel bubble sort!

\[ C(n) = \frac{n(n-1)}{2} \]
\[ D(n) = 2n - 3 \]
\[ \text{Cost} \times \text{Delay} = \Theta(n^3) \]

Fig. 7.8 Sorting network based on insertion sort or selection sort.
Theoretically Optimal Sorting Networks

AKS sorting network
(Ajtai, Komlos, Szemeredi: 1983)

The outputs are a permutation of the inputs satisfying
\[ y_0 \leq y_1 \leq \ldots \leq y_{n-1} \]

Note that even for these optimal networks, delay-cost product is suboptimal; but this is the best we can do.

Existing sorting networks have \( O(\log^2 n) \) latency and \( O(n \log^2 n) \) cost.

Given that \( \log_2 n \) is only 20 for \( n = 1,000,000 \), the latter are more practical.

Unfortunately, AKS networks are not practical owing to large (4-digit) constant factors involved; improvements since 1983 not enough.
7.4 Batcher Sorting Networks

Fig. 7.9 Batcher’s even–odd merging network for 4 + 7 inputs.
Proof of Batcher’s Even-Odd Merge

Use the zero-one principle

Assume:

- \( x \) has \( k \) 0s
- \( y \) has \( k' \) 0s

\( v \) has \( k_{\text{even}} = \left\lceil k/2 \right\rceil + \left\lceil k'/2 \right\rceil \) 0s

\( w \) has \( k_{\text{odd}} = \left\lfloor k/2 \right\rfloor + \left\lfloor k'/2 \right\rfloor \) 0s

Case a: \( k_{\text{even}} = k_{\text{odd}} \)

\[
\begin{array}{cccccccccccc}
& & & & & & & & & & & \\
v & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
w & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Case b: \( k_{\text{even}} = k_{\text{odd}} + 1 \)

\[
\begin{array}{cccccccccccc}
& & & & & & & & & & & \\
v & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
w & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Case c: \( k_{\text{even}} = k_{\text{odd}} + 2 \)

\[
\begin{array}{cccccccccccc}
& & & & & & & & & & & \\
v & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
w & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Out of order
Batcher’s Even-Odd Merge Sorting

Batcher’s \((m, m)\) even-odd merger, for \(m\) a power of 2:

\[
C(m) = 2C(m/2) + m - 1 = (m - 1) + 2(m/2 - 1) + 4(m/4 - 1) + \ldots = m \log_2 m + 1
\]

\[
D(m) = D(m/2) + 1 = \log_2 m + 1
\]

Cost \(\times\) Delay = \(\Theta(m \log^2 m)\)

Batcher sorting networks based on the even-odd merge technique:

\[
C(n) = 2C(n/2) + (n/2)(\log_2(n/2)) + 1 \approx n(\log_2 n)^2/2
\]

\[
D(n) = D(n/2) + \log_2(n/2) + 1 = D(n/2) + \log_2 n = \log_2 n (\log_2 n + 1)/2
\]

Cost \(\times\) Delay = \(\Theta(n \log^4 n)\)

Fig. 7.10 The recursive structure of Batcher’s even–odd merge sorting network.
Example Batcher’s Even-Odd 8-Sorter

Fig. 7.11  Batcher’s even-odd merge sorting network for eight inputs.
Bitonic-Sequence Sorter

Bitonic sequence:

1 3 3 4 6 6 6 2 2 1 0 0
Rises, then falls

8 7 7 6 6 6 5 4 6 8 8 9
Falls, then rises

8 9 8 7 7 6 6 6 5 4 6 8
The previous sequence, right-rotated by 2

Fig. 14.2 Sorting a bitonic sequence on a linear array.
Batcher’s Bitonic Sorting Networks

Fig. 7.12  The recursive structure of Batcher’s bitonic sorting network.

Fig. 7.13  Batcher’s bitonic sorting network for eight inputs.
7.5 Other Classes of Sorting Networks

Desirable properties:

a. Regular / modular (easier VLSI layout).
b. Simpler circuits via reusing the blocks.
c. With an extra block tolerates some faults (missed exchanges).
d. With 2 extra blocks provides tolerance to single faults (a missed or incorrect exchange).
e. Multiple passes through faulty network (graceful degradation).
f. Single-block design becomes fault-tolerant by using an extra stage.

Fig. 7.14 Periodic balanced sorting network for eight inputs.
Shearsort-Based Sorting Networks (1)

Fig. 7.15  Design of an 8-sorter based on shearsort on $2 \times 4$ mesh.
Shearsort-Based Sorting Networks (2)

Fig. 7.16 Design of an 8-sorter based on shearsort on 2×4 mesh.
7.6 Selection Networks

Direct design may yield simpler/faster selection networks

3rd smallest element

Can remove this block if smallest three inputs needed

Can remove these four comparators

Deriving an (8, 3)-selector from Batcher’s even-odd merge 8-sorter.
Categories of Selection Networks

Unfortunately we know even less about selection networks than we do about sorting networks.

One can define three selection problems [Knut81]:

I. Select the $k$ smallest values; present in sorted order
II. Select $k$th smallest value
III. Select the $k$ smallest values; present in any order

Circuit and time complexity: (I) hardest, (III) easiest
Type-III Selection Networks

Figure 7.17 A type III (8, 4)-selector.
**Classifier Networks**

**Classifiers:** Selectors that separate the smaller half of values from the larger half

**Use of classifiers for building sorting networks**

**Problem:** Given $O(\log n)$-time and $O(n \log n)$-cost $n$-classifier designs, what are the delay and cost of the resulting sorting network?
8A Search Acceleration Circuits

Much of sorting is done to facilitate/accelerate searching
- Simple search can be speeded up via special circuits
- More complicated searches: range, approximate-match

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>8A.1 Systolic Priority Queues</td>
</tr>
<tr>
<td>8A.2 Searching and Dictionary Operations</td>
</tr>
<tr>
<td>8A.3 Tree-Structured Dictionary Machines</td>
</tr>
<tr>
<td>8A.4 Associative Memories</td>
</tr>
<tr>
<td>8A.5 Associative Processors</td>
</tr>
<tr>
<td>8A.6 VLSI Trade-offs in Search Processors</td>
</tr>
</tbody>
</table>
8A.1 Systolic Priority Queues

**Problem:** We want to maintain a large list of keys, so that we can add new keys into it (insert operation) and obtain the smallest key (extract operation) whenever desired.

Unsorted list: Constant-time insertion / Linear-time extraction

Sorted list: Linear-time insertion / Constant-time extraction

Can both insert and extract operations (priority-queue operations) be performed in constant time, independent of the size of the list?
First Attempt: Via a Linear-Array Sorter

Insertion of new keys and read-out of the smallest key value can be done in constant time, but the “hole” created by the extracted value cannot be filled in constant time.

Fig. 2.9
Operating on every other clock cycle, allows holes to be filled

A Viable Systolic Priority Queue
Systolic Data Structures

Each node holds the smallest (S), median (M), and largest (L) value in its subtree.

Each subtree is balanced or has one fewer element on the left (root flag shows this).

Example: 20 elements, 3 in root, 8 on left, and 9 on right.

8 elements:

3 + 2 + 3

Update/access examples for the systolic data structure of Fig. 8.3.

- Insert 2
- Insert 20
- Insert 127
- Insert 195
- Extractmin
- Extractmed
- Extractmax
8A.2 Searching and Dictionary Operations

Parallel \((p + 1)\)-ary search on PRAM

\[
\log_{p+1}(n + 1) = \frac{\log_2(n + 1)}{\log_2(p + 1)} = \Theta(\log n / \log p) \text{ steps}
\]

Speedup \(\approx \log p\)

Optimal: no comparison-based search algorithm can be faster

A single search in a sorted list can’t be significantly speeded up through parallel processing, but all hope is not lost:

- Dynamic data (sorting overhead)
- Batch searching (multiple lookups)
### Dictionary Operations

<table>
<thead>
<tr>
<th>Basic dictionary operations: record keys $x_0, x_1, \ldots, x_{n-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>search</strong>($y$)</td>
</tr>
<tr>
<td><strong>insert</strong>($y, z$)</td>
</tr>
<tr>
<td><strong>delete</strong>($y$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Some or all of the following operations might also be of interest:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>findmin</strong></td>
</tr>
<tr>
<td><strong>findmax</strong></td>
</tr>
<tr>
<td><strong>findmed</strong></td>
</tr>
<tr>
<td><strong>findbest</strong>($y$)</td>
</tr>
<tr>
<td><strong>findnext</strong>($y$)</td>
</tr>
<tr>
<td><strong>findprev</strong>($y$)</td>
</tr>
<tr>
<td><strong>extractmin</strong></td>
</tr>
<tr>
<td><strong>extractmax</strong></td>
</tr>
<tr>
<td><strong>extractmed</strong></td>
</tr>
</tbody>
</table>

**Priority queue operations:** $\textit{findmin}$, $\textit{extractmin}$ (or $\textit{findmax}$, $\textit{extractmax}$)
8A.3 Tree-Structured Dictionary Machines

Combining in the triangular nodes

- **search(y):** Pass OR of match signals & data from “yes” side
- **findmin / findmax:** Pass smaller/larger of two keys & data
- **findmed:** Not supported here
- **findbest(y):** Pass the larger of two match-degree indicators along with associated record

Fig. 8.1 A tree-structured dictionary machine.
Insertion and Deletion in the Tree Machine

Counters keep track of the vacancies in each subtree.

Insertion: insert(y,z)

Deletion needs second pass to update the vacancy counters.

Redundant insertion (update?) and deletion (no-op?)

Implementation: Merge the circle and triangle trees by folding.

Figure 8.2 Tree machine storing 5 records and containing 3 free slots.
Physical Realization of a Tree Machine

Tree machine in folded form
VLSI Layout of a Tree

H-tree layout (used, e.g., for clock distribution network in high-performance microchips)
8A.4 Associative Memories

Associative or content-addressable memories (AMs, CAMs)
Binary (BCAM) vs. ternary (TCAM)

Mismatch in cell connects the match line ($ml$) to ground
If all cells in the word match the input pattern, a word match is indicated

Image source: http://www.pagiamtzis.com/cam/camintro.html
Word Match Circuitry

The match line is precharged and then pulled down by any mismatch

Image source: http://www.pagiamtzis.com/cam/camintro.html

Note that each CAM cell is nearly twice as complex as an SRAM cell
More transistors, more wires
CAM Array Operation

Figure 1: NOR-based CAM architecture (adapted from [Sch96])

Image source: http://www.pagiamtzis.com/cam/camintro.html
Current CAM Applications

Packet forwarding
Routing tables specify the path to be taken by matching an incoming destination address with stored address prefixes
Prefixes must be stored in order of decreasing length (difficult updating)

Packet classification
Determine packet category based on information in multiple fields
Different classes of packets may be treated differently

Associative caches / TLBs
Main processor caches are usually not fully associative (too large)
Smaller specialized caches and TLBs benefit from full associativity

Data compression
Frequently used substrings are identified and replaced by short codes
Substring matching is accelerated by CAM
History of Associative Processing

### Associative Memory
- Parallel masked search of all words
- Bit-serial implementation with RAM

### Associative Processor
- Add more processing logic to PEs

---

#### Table 4.1 Entering the second half-century of associative processing

<table>
<thead>
<tr>
<th>Decade</th>
<th>Events and Advances</th>
<th>Technology</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1940s</td>
<td>Formulation of need &amp; concept</td>
<td>Relays</td>
<td>Mega-bit-OPS</td>
</tr>
<tr>
<td>1950s</td>
<td>Emergence of cell technologies</td>
<td>Magnetic, Cryogenic</td>
<td>Giga-bit-OPS</td>
</tr>
<tr>
<td>1960s</td>
<td>Introduction of basic architectures</td>
<td>Transistors</td>
<td>Tera-bit-OPS</td>
</tr>
<tr>
<td>1970s</td>
<td>Commercialization &amp; applications</td>
<td>ICs</td>
<td></td>
</tr>
<tr>
<td>1980s</td>
<td>Focus on system/software issues</td>
<td>VLSI</td>
<td></td>
</tr>
<tr>
<td>1990s</td>
<td>Scalable &amp; flexible architectures</td>
<td>ULSI, WSI</td>
<td>Peta-bit-OPS</td>
</tr>
</tbody>
</table>
8A.5 Associative Processors

Associative or content-addressable memories-processors constituted early forms of SIMD parallel processing.

Fig. 23.1
Functional view of an associative memory/processor.
Search Functions in Associative Devices

*Exact match*: Locating data based on partial knowledge of contents

*Inexact match*: Finding numerically or logically proximate values

*Membership*: Identifying all members of a specified set

*Relational*: Determining values that are less than, less than or equal, etc.

*Interval*: Marking items that are between or outside given limits

*Extrema*: Finding the maximum, minimum, next higher, or next lower

*Rank-based*: Selecting $k$th or $k$ largest/smallest elements

*Ordered retrieval*: Repeated max- or min-finding with elimination (sorting)
Classification of Associative Devices

Handling of bits within words

<table>
<thead>
<tr>
<th>Parallel</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPBP: Fully parallel</td>
<td>WPBS: Bit-serial</td>
</tr>
<tr>
<td>WSBP: Word-serial</td>
<td>WSBS: Fully serial</td>
</tr>
</tbody>
</table>

Handling of words

Parallel

Serial
WSBP: Word-Serial Associative Devices

Strictly speaking, this is not a parallel processor, but with superhigh-speed shift registers and deeply pipelined processing logic, it behaves like one.
WPBS: Bit-Serial Associative Devices

One bit of every word is processed in one device cycle

Advantages:
1. Can be implemented with conventional memory
2. Easy to add other capabilities beyond search

Example: Adding field A to field B in every word, storing the sum in field S

Loop:
- Read next bit slice of A
- Read next bit slice of B
  (carry from previous slice is in PE flag C)
- Find sum bits; store in next bit slice of S
- Find new carries; store in PE flag

Endloop
Goodyear STARAN Associative Processor

First computer based on associative memory (1972)

Aimed at air traffic control applications

Aircraft conflict detection is an $O(n^2)$ operation

AM can do it in $O(n)$ time
Flip Network Permutations in the Goodyear STARAN

The 256 bits in a bit-slice could be routed to 256 PEs in different arrangements (permutations)

Figs. in this slide from J. Potter, “The STARAN Architecture and Its Applications …,” 1978 NCC
Distributed Array Processor (DAP)

Fig. 23.6
The bit-serial processor of DAP.
DAP’s High-Level Structure

- Program memory
- Master control unit
- Host interface unit
- Host workstation
- Array memory (at least 32K planes)
- Local memory for processor $ij$
- Register $Q$ in processor $ij$
- Fast I/O

Fig. 23.7 The high-level architecture of DAP system.
8A.6 VLSI Trade-offs in Search Processors

This section has not been written yet

References:


Many parallel processing techniques originate from, or find applications in, designing high-speed arithmetic circuits

- Counting, addition/subtraction, multiplication, division
- Limits on performance and various VLSI trade-offs

Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B.1</td>
<td>Basic Addition and Counting</td>
</tr>
<tr>
<td>8B.2</td>
<td>Circuits for Parallel Counting</td>
</tr>
<tr>
<td>8B.3</td>
<td>Addition as a Prefix Computation</td>
</tr>
<tr>
<td>8B.4</td>
<td>Parallel Prefix Networks</td>
</tr>
<tr>
<td>8B.5</td>
<td>Multiplication and Squaring Circuits</td>
</tr>
<tr>
<td>8B.6</td>
<td>Division and Square-Rooting Circuits</td>
</tr>
</tbody>
</table>
8B.1 Basic Addition and Counting

Using full-adders in building bit-serial and ripple-carry adders.

(a) Bit-serial adder.

Ideal latency: $O(\log k)$

Ideal cost: $O(k)$

Can these be achieved simultaneously?

(b) Ripple-carry adder.
Constant-Time Counters

Any fast adder design can be specialized and optimized to yield a fast counter (carry-lookahead, carry-skip, etc.)

One can use redundant representation to build a constant-time counter, but a conversion penalty must be paid during read-out.

Count register divided into three stages

Fig. 5.12 (in *Computer Arithmetic*)
Fast (constant-time) three-stage up counter.

Counting is fundamentally simpler than addition.
8B.2 Circuits for Parallel Counting

1-bit full-adder = (3; 2)-counter

Circuit reducing 7 bits to their 3-bit sum = (7; 3)-counter

Circuit reducing $n$ bits to their $\lceil \log_2(n + 1) \rceil$-bit sum = $(n; \lceil \log_2(n + 1) \rceil)$-counter

Fig. 8.16 (in *Computer Arithmetic*)
A 10-input parallel counter also known as a (10; 4)-counter.
Accumulative Parallel Counters

True generalization of sequential counters

Possible application: Compare Hamming weight of a vector to a constant

Latency: $O(\log n)$
Cost: $O(n)$
8B.3 Addition as a Prefix Computation

Example: Prefix sums

\[
\begin{align*}
  x_0 & \quad x_1 & \quad x_2 & \quad \ldots & \quad x_i \\
  x_0 & \quad x_0 + x_1 & \quad x_0 + x_1 + x_2 & \quad \ldots & \quad x_0 + x_1 + \ldots + x_i \\
  s_0 & \quad s_1 & \quad s_2 & \quad \ldots & \quad s_i
\end{align*}
\]

Sequential time with one processor is \( O(n) \)
Simple pipelining does not help

Fig. 8.4 Prefix computation using a latched or pipelined function unit.
Improving the Performance with Pipelining

Ignoring pipelining overhead, it appears that we have achieved a speedup of 4 with 3 “processors.” Can you explain this anomaly? (Problem 8.6a)

Fig. 8.5 High-throughput prefix computation using a pipelined function unit.
Carry Determination as a Prefix Computation

<table>
<thead>
<tr>
<th>$g_i$</th>
<th>$p_i$</th>
<th>Carry is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
<td>annihilated or killed</td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td>propagated</td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td>generated</td>
</tr>
<tr>
<td>1 1</td>
<td></td>
<td>(impossible)</td>
</tr>
</tbody>
</table>

$g_i = x_i \cdot y_i$

$p_i = x_i \oplus y_i$

Figure from *Computer Arithmetic*

Fig. 5.15 (ripple-carry network) superimposed on Fig. 5.14 (generic adder).
8B.4 Parallel Prefix Networks

This is the Brent-Kung Parallel prefix network (its delay is actually $2 \log_2 n - 2$)

$$T(n) = T(n/2) + 2 = 2 \log_2 n - 1$$

$$C(n) = C(n/2) + n - 1 = 2n - 2 - \log_2 n$$

Fig. 8.6 Prefix sum network built of one $n/2$-input network and $n - 1$ adders.
Example of Brent-Kung Parallel Prefix Network

Originally developed by Brent and Kung as part of a VLSI-friendly carry lookahead adder

One level of latency

\[
T(n) = 2 \log_2 n - 2
\]

\[
C(n) = 2n - 2 - \log_2 n
\]

Fig. 8.8 Brent–Kung parallel prefix graph for \( n = 16 \).
Another Divide-and-Conquer Design

Ladner-Fischer construction

\[ T(n) = T(n/2) + 1 \]
\[ = \log_2 n \]
\[ C(n) = 2C(n/2) + n/2 \]
\[ = (n/2) \log_2 n \]

Simple Ladner-Fisher Parallel prefix network
(its delay is optimal, but has fan-out issues if implemented directly)

Fig. 8.7 Prefix sum network built of two \( n/2 \)-input networks and \( n/2 \) adders.
Example of Kogge-Stone Parallel Prefix Network

Fig. 8.9 Kogge-Stone parallel prefix graph for $n = 16$.

$T(n) = \log_2 n$

$C(n) = (n-1) + (n-2) + (n-4) + \ldots + n/2$

$= n \log_2 n - n - 1$

Optimal in delay, but too complex in number of cells and wiring pattern.
Comparison and Hybrid Parallel Prefix Networks

Fig. 8.10 A hybrid Brent–Kung / Kogge–Stone parallel prefix graph for \( n = 16 \).

Brent/Kung
6 levels
26 cells

Kogge/Stone
4 levels
49 cells

Han/Carlson
5 levels
32 cells
Linear-Cost, Optimal Ladner-Fischer Networks

Define a type-$x$ parallel prefix network as one that:
- Produces the leftmost output in optimal $\log_2 n$ time
- Yields all other outputs with at most $x$ additional delay

Note that even the Brent-Kung network produces the leftmost output in optimal time.

We are interested in building a type-0 overall network, but can use type-$x$ networks ($x > 0$) as component parts.

Recursive construction of the fastest possible parallel prefix network (type-0)
Examples of Type-0, 1, 2 Parallel Prefix Networks

Fig. 8.10 A hybrid Brent–Kung / Kogge–Stone parallel prefix graph for $n = 16$. 

Brent/Kung: 16-input type-2 network

Kogge/Stone 16-input type-0 network

Han/Carlson 16-input type-1 network
8B.5 Multiplication and Squaring Circuits

Notation for our discussion of multiplication algorithms:

- **a** Multiplicand
- **x** Multiplier
- **p** Product \((a \times x)\)

Initially, we assume unsigned operands

Sequential:
- \(O(k)\) circuit complexity
- \(O(k)\) time with carry-save additions

Parallel:
- \(O(k^2)\) circuit complexity
- \(O(\log k)\) time

**Fig. 9.1** (in *Computer Arithmetic*) Multiplication of 4-bit binary numbers.
Divide-and-Conquer (Recursive) Multipliers

Building wide multiplier from narrower ones

\[
\begin{array}{c}
\text{a}_H \\
\times \\
\text{x}_H \\
\hline \\
\text{a}_L \\
\text{x}_L \\
\hline \\
\text{a}_L \text{x}_L \\
\text{a}_L \text{x}_H \\
\text{a}_H \text{x}_L \\
\hline \\
\text{a}_H \text{x}_H \\
\hline \\
p
\end{array}
\]

Rearranged partial products in 2b-by-2b multiplication

\[
\begin{array}{c}
\text{2b bits} \\
\hline \\
\text{b bits} \\
\hline \\
\text{3b bits}
\end{array}
\]

Fig. 12.1 (in Computer Arithmetic) Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ multiplier from $b \times b$ multipliers.

\[
C(k) = 4C(k/2) + O(k) = O(k^2)
\]

\[
T(k) = T(k/2) + O(\log k) = O(\log^2 k)
\]
(Anatoly) Karatsuba Multiplication

$2b \times 2b$ multiplication requires four $b \times b$ multiplications:

$$(2^b a_H + a_L) \times (2^b x_H + x_L) = 2^{2b} a_H x_H + 2^b (a_H x_L + a_L x_H) + a_L x_L$$

Karatsuba noted that one of the four multiplications can be removed at the expense of introducing a few additions:

$$(2^b a_H + a_L) \times (2^b x_H + x_L) =$$

$$2^{2b} a_H x_H + 2^b [(a_H + a_L) \times (x_H + x_L) - a_H x_H - a_L x_L] + a_L x_L$$

Benefit is quite significant for extremely wide operands

$$C(k) = 3C(k/2) + O(k) = O(k^{1.585})$$

$$T(k) = T(k/2) + O(\log k) = O(\log^2 k)$$
Divide-and-Conquer Squarers

Building wide squarers from narrower ones

Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ squarer from $b \times b$ squarers and multiplier.
VLSI Complexity Issues and Bounds

Any VLSI circuit computing the product of two $k$-bit integers must satisfy the following constraints:

\[ AT \] grows at least as fast as $k^{3/2}$
\[ AT^2 \] is at least proportional to $k^2$

Array multipliers: $O(k^2)$ gate count and area, $O(k)$ time

\[ AT = O(k^3) \quad \text{and} \quad AT^2 = O(k^4) \]

Simple recursive multipliers: $O(k^2)$ gate count, $O(\log^2 k)$ time

\[ AT = O(k^2 \log^2 k) \quad \text{and} \quad AT^2 = O(k^2 \log^4 k) \]

Karatsuba multipliers: $O(k^{1.585})$ gate count, $O(\log^2 k)$ time

\[ AT = O(k^{1.585} \log^2 k) \quad \text{and} \quad AT^2 = O(k^{1.585} \log^4 k) \]

Discrepancy due to the fact that interconnect area is not taken into account in our previous analyses.
Theoretically Best Multipliers

Arnold Schonhage and Volker Strassen (via FFT); best until 2007

- O(log $k$) time
- O($k \log k \log \log k$) complexity

In 2007, Martin Furer managed to replace the log log $k$ term with an asymptotically smaller term (for astronomically large numbers)

It is an open problem whether there exist logarithmic-delay multipliers with linear cost
(it is widely believed that there are not)

In the absence of a linear cost multiplication circuit, multiplication must be viewed as a more difficult problem than addition

In 2019, David Harvey and Joris van der Hoeven developed an $O(n \log n)$ multiplication algorithm, which is believed to be the best possible theoretically (but not practical at present)
8B.6 Division and Square-Rooting Circuits

Division via Newton’s method: $O(\log k)$ multiplications

Using Schonhage and Strassen’s FFT-based multiplication, leads to:

$O(\log^2 k)$ time
$O(k \log k \log \log k)$ complexity

With the multiplication algorithm of Harvey and van der Hoeven:

$O(\log^2 k)$ time
$O(k \log k)$ complexity
Theoretically Best Dividers

Best known bounds; cannot be achieved at the same time (yet)

- $O(\log k)$ time
- $O(k \log k)$ complexity

In 1966, S. A. Cook established these simultaneous bounds:

- $O(\log^2 k)$ time
- $O(k \log k \log \log k)$ complexity

In 1983, J. H. Reif reduced the time complexity to the current best

- $O(\log k (\log \log k)^2)$ time

In 1984, Beame/Cook/Hoover established these simultaneous bounds:

- $O(\log k)$ time
- $O(k^4)$ complexity

Given our current state of knowledge, division must be viewed as a more difficult problem than multiplication
Implications for Ultrawide High-Radix Arithmetic

Arithmetic results with $k$-bit binary operands hold with no change when the $k$ bits are processed as $g$ radix-$2^h$ digits ($gh = k$)
Another Circuit Model: Artificial Neural Nets

**Feedforward network**
Three layers: input, hidden, output
No feedback

**Recurrent network**
Simple version due to Elman
Feedback from hidden nodes to special nodes at the input layer

**Hopfield network**
All connections are bidirectional

Characterized by connection topology and learning method

Diagrams from http://www.learnartificialneuralnetworks.com/
8C Fourier Transform Circuits

Fourier transform is quite important, and it also serves as a template for other types of arithmetic-intensive computations
  • FFT; properties that allow efficient implementation
  • General methods of mapping flow graphs to hardware

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>8C.1 The Discrete Fourier Transform</td>
</tr>
<tr>
<td>8C.2 Fast Fourier Transform (FFT)</td>
</tr>
<tr>
<td>8C.3 The Butterfly FFT Network</td>
</tr>
<tr>
<td>8C.4 Mapping of Flow Graphs to Hardware</td>
</tr>
<tr>
<td>8C.5 The Shuffle-Exchange Network</td>
</tr>
<tr>
<td>8C.6 Other Mappings of the FFT Flow Graph</td>
</tr>
</tbody>
</table>
8C.1 The Discrete Fourier Transform

Other important transforms for discrete signals:
- z-transform (generalized form of Fourier transform)
- Discrete cosine transform (used in JPEG image compression)
- Haar transform (a wavelet transform, which like DFT, has a fast version)

Some operations are easier in frequency domain; hence the need for transform.
Defining the DFT and Inverse DFT

DFT yields output sequence $y_i$ based on input sequence $x_i \ (0 \leq i < n)$

$$y_i = \sum_{j=0}^{n-1} \omega_n^{ij} x_j \quad \text{O}(n^2)\text{-time naïve algorithm}$$

where $\omega_n$ is the $n$th primitive root of unity; $\omega_n^n = 1, \ \omega_n^j \neq 1 \ (1 \leq j < n)$

Examples:

$$\omega_4 = i$$
$$\omega_3 = (\frac{-1 + i\sqrt{3}}{2})$$
$$\omega_8 = \sqrt{2}(1 + i)/2$$

The inverse DFT is almost exactly the same computation:

$$x_i = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{-ij} y_j$$

Input seq. $x_i \ (0 \leq i < n)$ is said to be in time domain

Output seq. $y_i \ (0 \leq i < n)$ is the input’s frequency-domain characterization
DFT of a Cosine Waveform

DFT of a cosine with a frequency 1/10 the sampling frequency $f_s$
DFT of a Cosine with Varying Resolutions

DFT of a cosine with a frequency 1/10 the sampling frequency $f_s$
DFT as Vector-Matrix Multiplication

DFT and inverse DFT computable via matrix-by-vector multiplication

\[ y_i = \sum_{j=0}^{n-1} \omega_{n}^{ij} x_j \]

DFT matrix

\[
W = \frac{1}{\sqrt{N}} \begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)}
\end{bmatrix}
\]
DFT Basics and Visualizations

Fourier transform, Fourier series, and frequency spectrum (16-min. video)
https://www.youtube.com/watch?v=r18Gi8lSkfM

Discrete Fourier transform: Introduction (11-minute video)
https://www.youtube.com/watch?v=mkGsMWi_j4Q

A visual introduction to Fourier transform (21-minute video)
https://www.youtube.com/watch?v=spUNpyF58BY
Application of DFT to Smoothing or Filtering

Input signal with noise → DFT → Low-pass filter → Inverse DFT →Recovered smooth signal
DFT Application Example

Signal corrupted by 0-mean random noise

FFT shows strong frequency components of 50 and 120

The uncorrupted signal was:

\[ x = 0.7 \sin(2\pi \cdot 50t) + \sin(2\pi \cdot 120t) \]

Source of images:
Application of DFT to Spectral Analysis

Tone frequency assignments for touch-tone dialing

Frequency spectrum of received tone
8C.2 Fast Fourier Transform

DFT yields output sequence $y_i$ based on input sequence $x_i$ ($0 \leq i < n$)

$$y_i = \sum_{j=0}^{n-1} \omega_n^{ij} x_j$$

**Fast Fourier Transform (FFT):**

The Cooley-Tukey algorithm

$O(n \log n)$-time DFT algorithm that derives $y$ from half-length sequences $u$ and $v$ that are DFTs of even- and odd-indexed inputs, respectively

$$y_i = u_i + \omega_n^i v_i \quad (0 \leq i < n/2)$$

$$y_{i+n/2} = u_i + \omega_n^{i+n/2} v_i = u_i - \omega_n^i v_i$$

Butterfly operation

$$T(n) = 2T(n/2) + n = n \log_2 n$$

sequentially

$$T(n) = T(n/2) + 1 = \log_2 n$$

in parallel
More General Factoring-Based Algorithm

1d DFT of size $N$:

\[ N = N_1 N_2 \]

\[ \approx \text{2d DFT of size } N_1 \times N_2 \]

*inputs:*

- Interpret 1d inputs:
- Multiply by $N$ “twiddle factors”
- Transpose

\[ = \text{contiguous} \]

*first DFT columns, size } N_2 \]
\[ \text{(non-contiguous)} \]

*finally, DFT columns, size } N_1 \]
\[ \text{(non-contiguous)} \]
8C.3 The Butterfly FFT Network

\[ y_i = u_i + \omega_n^i v_i \quad (0 \leq i < n/2) \]
\[ y_{i+n/2} = u_i + \omega_n^{i+n/2} v_i \]

**Fig. 8.11** Butterfly network for an 8-point FFT.

- **u**: DFT of even-indexed inputs
- **v**: DFT of odd-indexed inputs
Butterfly Processor

Performs a pair of multiply-add operations, where the multiplication is by a constant.

Design can be optimized by merging the adder and subtractor, as they receive the same inputs.
Computation Scheme for 16-Point FFT

Bit-reversal permutation

Butterfly operation
\[ a, b \rightarrow a + b \omega^j \]
\[ a, b \rightarrow a - b \omega^j \]
8C.4 Mapping of Flow Graphs to Hardware

Given a computation flow graph, it can be mapped to hardware

Fig. 25.6 Parhami’s textbook on computer arithmetic.
Ad-hoc Scheduling on a Given Set of Resources

Given a computation flow graph, it can be mapped to hardware

Assume:
\[ t_{\text{add}} = 1 \]
\[ t_{\text{mult}} = 3 \]
\[ t_{\text{div}} = 8 \]
\[ t_{\text{sqrt}} = 10 \]
Mapping through Projection

Given a flow graph, it can be projected in various directions to obtain corresponding hardware realizations.

Multiple nodes of a flow graph may map onto a single hardware node.

That one hardware node then performs the computations associated with the flow graph nodes one by one, according to some timing arrangement (schedule).

Linear array, with each cell acting for one butterfly network row.
8C.5 The Shuffle-Exchange Network
Variants of the Butterfly Architecture

Fig. 8.12  FFT network variant and its shared-hardware realization.
8C.6 Other Mappings of the FFT Flow Graph

This section is incomplete at this time
More Economical FFT Hardware

Fig. 8.13 Linear array of $\log_2 n$ cells for $n$-point FFT computation.
Space-Time Diagram for the Feedback FFT Array

Feedback butterfly processor